One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

fido5100BBCZ and fido5200BBCZ Real-Time Ethernet Multiprotocol Switch and fido2100 3-Port Industrial Ethernet DLR Switch with IEEE 1588

FEATURES

DLR porting layer support code
fido2100 DLR switch interrupt event handling
DLR packet processing
DLR protocol implementation
Ethernet/IP DLR object support

GENERAL DESCRIPTION

The DLR support library provides all the software required to support the beacon-based device level ring (DLR) protocol using either the Analog Devices, Inc. fido5100BBCZ and fido5200BBCZ real-time Ethernet multiprotocol (REM) switch or the Analog Devices fido2100 DLR switch. To complete an Ethernet/IP device, this library must be combined with a TCP/IP stack and an Ethernet/IP stack. Both of these stacks must be provided by the user. When using the fido5100BBCZ and fido5200BBCZ REM switch, it is also necessary to include the Innovasic REM driver, which is available separately on the DET Developer Portal.

Figure 1 shows the relationship of these functional areas, and the DLR Porting Layer Support Code section through the DLR Object Support section describe each of these functional areas in more detail.

This library only supports creating a DLR node and not a DLR supervisor.

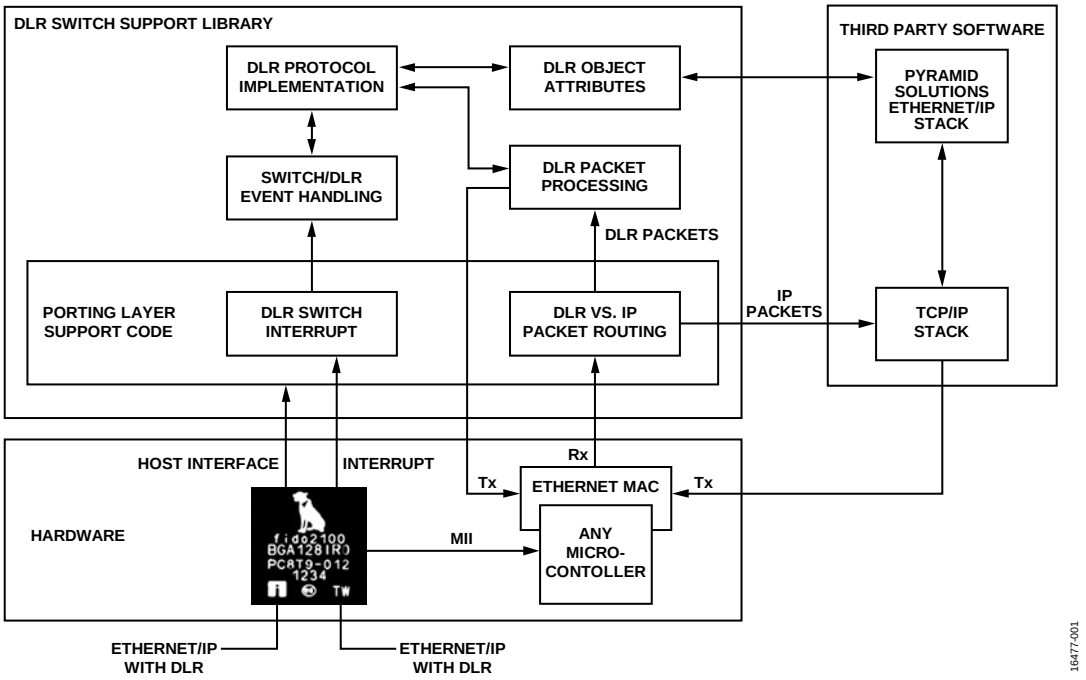

Figure 1. DLR Switch Library Overview

TABLE OF CONTENTS

REVISION HISTORY

This Innovasic product user guide has been reformatted to the styles and standards of Analog Devices, Inc.

5/2018—Rev. 03 to Rev. D

INTRODUCTION

DLR PORTING LAYER SUPPORT CODE

The library contains a set of board support package files that collocate all the implementation and low level (hardware) access functions. These files require some work because the user uses these files to connect this library to the design-specific real-time operating system (RTOS) and hardware. The initialization process of the DLR library also makes use of these board support package (BSP) and porting layer functions so that after the BSP and porting layer functions are in place, the DLR library is self-contained. The Porting Layer Detailed Description section discusses the porting layer in more detail.

This DLR support library is used in a system that makes use of a real-time operating system.

fido2100 DLR SWITCH INTERRUPT EVENT HANDLING

The library provides a handler function that contains all the necessary code to manage the hardware interrupt events that the fido2100 DLR switch generates. All that is required is to connect this interrupt handler to the physical, low level hardware interrupt mechanism used by the microcontroller implemented in the design. After the connection is established, the DLR library takes care of any remaining requirements.

DLR PACKET PROCESSING

The DLR protocol makes use of several non-IP Ethernet packets. These packets must be routed to the packet handling code within this library. All other, non-DLR, network traffic must be routed to a TCP/IP or Ethernet/IP stack. Because it is impossible to anticipate the user hardware or TCP/IP stack requirements, this packet routing is the responsibility of the user. Other than this one requirement, the Innovasic Ethernet switch hardware and the DLR library manages all DLR operations relating to these special packets.

DLR PROTOCOL IMPLEMENTATION

The DLR library detects the various DLR events and provides all of the functionality required by the DLR protocol. These functionalities includes the sign on process; neighbor check request; response and status frame processing; ring fault and restoration processing; and beacon and announce frame management.

DLR OBJECT SUPPORT

The ODVA Ethernet/IP specification describes a set of DLR object attributes (both class and instance) that must be supported by all DLR aware devices. The DLR library provides a set of files that implement these attributes. The object supplied is used within the Pyramid Solutions Ethernet/IP stack, but the object is a standard code that can be ported to any other Ethernet/IP stack.

USING THE DLR LIBRARY

USE CASES

When creating an Ethernet/IP device using the Innovasic Ethernet switch hardware, there are four possible use cases. Each case has its own unique considerations. The essential differences are the compile environment, the libraries that are included, and how the hardware is initialized.

The use cases and the necessary build actions are as shown in Table 1.

Table 1. Actions Required for a Given Chip Depending on DLR Implementation

Use Case		Actions Required		
Switch in Use	**Is DLR Enabled?**	**REM Driver**	**DLR Library**	**Comments**
fido5100BBCZ/fido5200BBCZ	No	Yes	No	Requires no calls to any DLR library functions
fido5100BBCZ/fido5200BBCZ	Yes	Yes	Yes	Must enable DLR in EtherIpRingProtocol_Initialize
fido2100	No	No	Yes	Must disable DLR in EtherIpRingProtocol_Initialize
fido2100	Yes	No	Yes	Must enable DLR in EtherIpRingProtocol_Initialize

COMPILING AND LINKING

The REM Switch Driver User Guide describes the requirements for the build environment. The include directory requirements for the DLR library are as follows: the DLR library itself is flat, but the TCP/IP and Ethernet/IP stacks must be included in the search path.

Some of the files provided in the library include files provided by the sample processor libraries. These files must be removed and/or changed (see the Porting Layer Detailed Description section).

Table 2. Requirements Specific to Certain Use Cases

Requirement	Use Case
fido2100 (With or Without DLR)	Include only the DLR library source files and do not define the USE_REMS preprocessor symbol.
REM (With DLR)	Include all the source files from both the DLR library and the REMS driver, and define the USE_REMS preprocessor symbol.
REM (Without DLR)	Include only the REM library source files, and the definition of the USE_REMS preprocessor symbol is irrelevant (only the DLR uses USE_REMS).

ADJUSTMENTS FOR DIFFERENT HARDWARE PLATFORMS

The DLR library includes several files that are specifically designed for use with the sample processor on the RapID platform. These files must be modified to be used elsewhere if the user is using a different host processor. The files that must be changed are as follows:

- The **BspEnetSwitch.c/h** and **EtherIpRingRx.c/h** files are included to provide a complete and functional example. These files are specific to the platform they were developed on and are dependent on several of the sample processor library include files. These board support files must be modified for the actual hardware. See the Porting Layer Detailed Description section for details.
- The **EtherIpRingObject.c** file also contains many references to the sample processor library as well as the processor platform header files from the Pyramid stack. The sample processor headers must be replaced with the headers from the actual libraries and platform.
- The **EtherIpRingProtocol.c** file includes a header called **syslog.h** to use the sample processor syslog facility. This include directive can be safely deleted. No other changes are needed.

REM DRIVER DEPENDENCIES

As delivered, the DLR library has certain dependencies on the REM driver files. These dependencies can cause a problem if the fido2100 switch is used instead of the fido5100BBCZ/ fido5200BBCZ REM switch. The dependencies must be removed.

The dependencies are limited to two files: **BspEnetSwitch.c** and **EtherIpRingProtocol.c.** The first file contains a call to the function REMS_DlrXmitPacket, but this is replaced when the board support files are adapted to the actual platform in use. The second file contains calls to the following REM driver functions:

- REMS_DlrEnable
- REMS_FlushDynamicTable
- REMS_EipStartTcu
- REMS_EipEnableBTOIrq
- REMS_EipDisableBTOIrq

The DLR library isolates these calls to remove these dependencies. All that is necessary is to omit the definition of the USE_REMS preprocessor symbol when the library file is compiled.

LIBRARY FILE SET GENERAL DESCRIPTION

EtherIpRingUtils.h

The EtherIpRingUtils.h file contains a set of generic, static inline helper functions to read/write 16-, 32-, and 48-bit data to or from the fido2100 switch registers in big and little endian form. There are no changes required for any of these functions.

EtherIpRingRx.h/c

The code in the EtherIpRingRx.h/c files is written to utilize the Ethernet medium access control (MAC) on the sample processor and satisfies the packet routing process described herein. This file is not used with the fido5100BBCZ and fido5200BBCZ switch; however, the DLR packets must still be routed into the DLR library in the same manner. For this reason, it is necessary to have at least a basic understanding of the packet flow. The two functions defined in this section are as follows:

- FIDO_DLR_ethRXComplete()
- FIDO_ethGetDWORDAtPacketOffset()

The first function is a public function that contains the code to receive and route received Ethernet packets. The second function is a local static helper function to assist in the process. Although specific implementation provided in this section is not directly useful to the user, the implementation is provided to illustrate the processing required. The functions are provided as part of the library but are not called by this DLR library code. In the case of the sample processor, when an Ethernet packet is received, the interrupt handler calls the public function. The provided code is the way of doing this using the sample processor, but the user must provide code appropriate for the platform in use. The important thing is to receive and route DLR packets as described in this section. DLR packets are routed to this library and all others are routed to the TCP/IP stack.

The actual code provided is heavily commented and is not described further in this section. The user is free to develop code that is appropriate to the platform in use. The key tasks are as follows:

- Use EtherIpRingProtocol_Ready() (which is provided in the DLR library) to determine when the DLR library is ready for packets.
- Use whatever means necessary to examine the packet for the 802.1Q tag and Ethertype word and route the packets accordingly (DLR frames have a distinct Ethertype). Depending on the hardware, the packet examination is completed before or after the packet is read out of the Ethernet MAC. If using DMA, the packet examination likely must be done in the destination memory after the DMA is complete. If the Ethernet driver stack uses a scatter gather methodology, this examination may be completed after the packet is reassembled. All standard TCP/IP packets must be routed to the user TCP/IP stack in the manner expected by that component. Because of the wide variety of TCP/IP stacks that can be used, this is not described in this section.
- The BSP porting layer functions that are used after a packet is determined to be a DLR packet are as follows:
  - Obtain a memory pool buffer into which to read the received DLR packet data.
  - Release the buffer back to the pool on error.
  - Add the packet data to the DLR packet queue.
- After adding the DLR packet to the DLR packet queue, call EtherIpRingProtocol_ReceiveCallBack() (provided in the DLR library) to inform the DLR library.

The DLR packet is placed into the queue using a structure defined as follows:

```
typedef volatile struct {
  unsigned int size;
  char* data_p;
 } DLR_Buffer_t;
```

The structure contains two things: a pointer to the packet data and the packet size. One simple way to place the DLR packed into the queue is to ensure the buffers returned by the memory pool are large enough to contain the packet data as well as the structure contents. Then, the structure can be prepended to the packet data in a single memory buffer.

The following pseudo code shows the recommended flow of the process:

```
EthernetPacketReceiveHandler()
{
 DLR_Buffer_t* DLR_Packet_p; // struct to
describe DLR packet
 unsigned short pktTypeIsDLR = 0; // default
all packets to non-DLR
 unsigned int pktSize; // size of packet
copied from Ethernet MAC


 if (no errors and received packet is valid)
{
 if (EtherIpRingProtocol_Ready()) {
 if ((packet is 802.1q tagged) && (packet
has DLR Ethertype)) {
 pktTypeIsDLR = 1; // override default above
 }
 }


 if (pktTypeIsDLR == 0) {
 // this means the packet is NOT DLR
 // if present and if necessary remove the
802.1q tag
 // obtain a buffer from the TCP/IP stack
 // read the packet into this buffer
 // give packet to TCP/IP stack
```

```
 }
 else {
 // the packet IS DLR so we need to route it
to the DLR Library
 // use BSP porting layer to obtain a memory
pool buffer:
 DLR_Packet_p =
(DLR_Buffer_t*)BSP_MemPoolReserveBuffer(&g_D
LR_BufferPool);
 if (DLR_Packet_p != NULL) {
 // set up the packet pointer to memory past
the struct itself
 DLR_Packet_p->data_p =
(char*)(DLR_Packet_p) +
sizeof(DLR_Buffer_t);
 // copy data from MAC into memory at
DLR_Packet_p->data_p...
 if (copy or packet read error) {
 // on error free memory pool buffer and get
out
 BSP_MemPoolReleaseBuffer(&DLR_Packet_p);
 return;
 }
 // assume 'pktSize' is set by function used
to copy packet
 // use this variable to set the size in the
struct
 DLR_Packet_p->size = pktSize;


 // packet and struct is in memory pool,
feed the DLR Library
 BSP_QueueAdd(g_DLR_PacketQueue,
&DLR_Packet_p, NO_WAIT);


 // and tell the DLR Library main task to
process the packet
 EtherIpRingProtocol_ReceiveCallBack();
 }
 else {
 // unable to obtain a buffer from the
memory pool
 // silently discard the data...
 }
 }
 }
 else
 // packet error, dump it...
}
```

BspEnetSwitch.h/c

The BspEnetSwitch.h/c files contain an abstraction or porting layer that connects the DLR library to the RTOS as well as the central processing unit (CPU) and low level hardware. The user must implement a number of these functions to create a complete system. These functions are gathered in these files and facilitate integrating this library into the overall board support package in a system. Some of the functions in these files are implementation dependent. These functions are in this file despite being implementation dependent, because they are part of the low level fido2100 DLR switch hardware management. These functions require some user modification. The other functions are functional in their current form. Rather than providing empty stubs, the implementation dependent functions provided in the files contain the sample processor implementations for reference.

The implementation dependent board support functions do the following:

- Initialize the board support package
- Set up the CPU fido2100 DLR switch chip select hardware
- Enable and disable CPU interrupts
- Get the MAC address of the system
- Determine the link speed and duplex
- Trigger the transmission of a DLR packet

The RTOS porting layer functions are used for the following:

- Create tasks
- Create, take, and post semaphores
- Apply to the rest of the bullets
- Mutual exclusion objects
- Timers
- The memory pool
- Queues

There are several functions that are hardware related but do not require changes. These functions do the following:

- Retrieve and post integer-based events
- Retrieve a DLR packet from the received packet queue
- Add and delete multicast addresses to and from the fido2100 DLR switch hash table
- Manage work related to link up and down changes
- Manage the Ethernet link and port enable and disable
- Manage the fido2100 DLR switch counters
- Manage the lookup table and calculate the cyclic redundancy check (CRC) used by the fido2100 DLR switch multicast hash table
- Read and write fido2100 DLR switch registers

A detailed description of the porting layer is provided in the EtherIpRingProtocol.h/c section.

EtherIpRingProtocol.h/c

Primarily, the EtherIpRingProtocol.h/c files contain the functions needed to manage the switch hardware as it pertains to the actual DLR protocol. Also provided are functions used by the EtherIpRingObject to manage the Ethernet/IP DLR object. No platform specific functions are present in this file, so there are no user changes required.

An RTOS is required by this library, and the code for the required tasks (threads) is also present in this file. The tasks are as follows:

- EtherIpRingProtocol_LinkMonTask()
- EtherIpRingProtocol_MainTask()
- EtherIpRingProtocol_TimerTask()

When the user calls EtherIpRingProtocol_Initialize(), these tasks are automatically launched.

EtherIpRingObject.h/c

The EtherIpRingObject.h/c files contain the functions needed to manage the required Ethernet/IP DLR object. This code is used with the Pyramid Ethernet/IP stack. The two functions that the user code must call here are as follows:

- During system initialization, register the DLR object with the Pyramid stack using EtherIpRingObject_RegisterObject().
- Within the NM_CLIENT_OBJECT_REQUEST_RECEIVED case in the user call back function (called by the Pyramid Ethernet/IP stack), call EtherIpRingObject_ProcessRequest() when a request is made to the ETHERNETIP_DLR_OBJECT _CLASSID:

```
clientGetClientRequest(iRequestId,
&Request);
if (Request.iClass ==
ETHERNETIP_DLR_OBJECT_CLASSID)

EtherIpRingObject_ProcessRequest(iRequest
Id, pRequest);
else ... // if the request wasn't
directed to the DLR object...
```

No platform specific functions are present in this file, so there are no user changes required.

When using Ethernet/IP DLR, it is a requirement to also support the quality of service (QoS) object. Support for this object is available as a compile time option within the Pyramid Ethernet/IP stack. The user manages the support for the QoS object outside of this library.

If creating a device that is intended to be capable of acting as a DLR supervisor (only supported if using the fido2100), there are certain operating parameters that must be nonvolatile. In reference to the operating parameters, the **EtherIpRingProtocol.h** file contains a data structure declared as follows:

```
typedef struct _RingParams {
UINT32 m_ulBeaconInterval; // Beacon
interval, microseconds
UINT32 m_ulBeaconTimeout; // Beacon
timeout, microseconds
UINT16 m_uwVlanId; // VLAN ID, 0-4094
UINT8 m_ubSupervisor; // Is supervisor
enabled
UINT8 m_ubPrecedence; // Supervisor
precedence
} RingParams;
```

EtherIpRingProtocol.c declares a global instance of this structure called m_ConfigParams. In the supplied code, this variable is initialized within EtherIpRingProtocol_Initialize() but the existing code assigns these values from hard coded constants. These parameters must be stored in nonvolatile memory if or when they are changed. EtherIpRingProtocol_SetRingParams() is the place to store these parameters. Adding the code to store and restore this data from nonvolatile memory is the responsibility of the user.

PORTING LAYER DETAILED DESCRIPTION

The **BspEnetSwitch.h/c** files contain the porting layer declaration and definition. This section provides a detailed description of each of the functions in these files. The description is broken down into the following five areas:

- Implementation dependent board support porting layer functions
- Implementation dependent RTOS support porting layer functions
- Implementation independent DLR library functions
- Implementation independent static functions in the **.c** file
- Implementation independent static inline functions in the **.h** file

These functions are described in the Implementation Dependent Board Support Porting Layer Functions section through the Implementation Independent Static Inline Functions in the .h File section. **BspEnetSwitch.h** is richly commented and has detailed function descriptions describing the function arguments and return values.

IMPLEMENTATION DEPENDENT BOARD SUPPORT PORTING LAYER FUNCTIONS

This section describes the board support and RTOS porting layer functions in more detail. The user must implement many of these functions. Rather than providing empty stubs, the implementation dependent functions provided in these files contain the sample processor implementations for reference.

BSP_EsInitialize()

The BSP_EsInitialize() function initializes the board support package and (using the DLR library RTOS porting layer) creates a task, two queues, and the memory pool. Specifically, the BSP_EsInitialize() function does the following:

- Create the bsp_EsStatCountTask() task
- Create and initialize the DLR packet queue and memory pool
- Create the DLR library event queue

The BSP_EsInitialize() function does not require many changes but may need some attention to ensure successful integration with the user supplied RTOS porting layer functions.

There is no need for the user to directly call BSP_EsInitialize(), because it is indirectly called when the user initializes the DLR library using EthernetIpRingProtocol_Initialize(). The recommended initialization sequence is discussed in the Implementation Independent Static Inline Functions in the .h File section.

BSP_SetupChipSelect()

In the sample processor implementation, the BSP_SetupChipSelect() function sets up the chip select control and timing registers so the CPU can access the fido2100 DLR switch. For the sample processor, the BSP_SetupChipSelect() function is called as a part of the overall system initialization, not the initialization of the DLR library. Other processors may set up the chip select registers and chip control registers differently or may not need this function at all. If the function is not needed, this function can be simply removed or ignored.

BSP_DisableSwitchInterrupts() and BSP_EnableSwitchInterrupts()

The BSP_DisableSwitchInterrupts() function and the BSP_ EnableSwitchInterrupts() function disable and enable the CPU interrupts. As provided, these functions disable and enable the CPO interrupts using the sample processor HAL library. A user's system does not have the sample processor HAL library, so these functions must be changed accordingly.

BSP_GetMacAdrs()

The BSP_GetMacAdrs() returns the MAC address assigned to the system. In the case of the sample processor a global variable is read. The user must obtain the MAC address by whatever means is provided by that platform.

BSP_EsUpdateLinkSpeedDuplex()

The responsibility of the BSP_EsUpdateLinkSpeedDuplex() function is to determine the Ethernet link speed and duplex and to update the bsp_EsLinkData global data for the specified port. The provided example reads the registers in the physical layer (PHY) using code supplied by the sample processor Ethernet driver. The user version must provide its own methods to obtain this information. This function is called by EtherIpRingProtocol_ ProcessLinkState() just prior to calling BSP_EsLinkUp().

BSP_Put_DLR_Packet()

The BSP_Put_DLR_Packet() function is called by the DLR library anytime it must transmit a packet. BSP_Put_DLR_Packet() function uses the supplied information to form the packet and then uses the sample processor Ethernet driver functions to write the packet to the Ethernet transmission hardware FIFO. The part of this function that formulates the packet is useful as is, but the part where the packet is written to the hardware must be replaced by the appropriate method used by the user platform.

IMPLEMENTATION DEPENDENT RTOS SUPPORT PORTING LAYER FUNCTIONS

The implementation dependent RTOS support porting layer functions include the functions listed in this section See the BspEnetSwitch.h/c section for a detailed description of the arguments and expected return values.

The existing file contains the following fido specific code for reference only:

- BSP_CreateTask()
- BSP_SemaphoreCreate()
- BSP_SemaphoreWait()
- BSP_SemaphorePost()
- BSP_MutexCreate()
- BSP_MutexLock()
- BSP_MutexUnlock()
- BSP_GetTicksPerSec()
- BSP_TimerCreate()
- BSP_TimerWait()
- BSP_MemPoolCreate()
- BSP_MemPoolReserveBuffer()
- BSP_MemPoolReleaseBuffer()
- BSP_QueueCreate()
- BSP_QueuePeek()
- BSP_QueueRemove()
- BSP_QueueAdd()

IMPLEMENTATION INDEPENDENT DLR LIBRARY FUNCTIONS

BSP_GetEvent()

The BSP_GetEvent() uses the BSP porting layer queue facility to retrieve an integer-based event code from the DLR library event queue. Because this function uses the porting layer, it is implementation independent.

BSP_PostEvent()

The BSP_PostEvent() uses the BSP porting layer queue facility to post an integer-based event code to the DLR library event queue. Because this function uses the porting layer, it is implementation independent.

BSP_Get_DLR_Packet()

The BSP_Get_DLR_Packet() uses the BSP porting layer to get a DLR packet from the DLR packet queue, copy it into the buffer provided, and free the memory used by the memory pool. Because this function uses the porting layer, it is implementation independent.

BSP_EsMulticastAdrsAdd and BSP_EsMulticastAdrsDel

The BSP_EsMulticastAdrsAdd and BSP_EsMulticastAdrsDel functions add and delete (respectively) a multicast address from the fido2100 DLR switch hash table. These functions access the fido2100 DLR switch registers but are not implementation dependent.

BSP_EsLinkUp() and BSP_EsLinkDown()

The BSP_EsLinkUp() and BSP_EsLinkDown()functions are called by EtherIpRingProtocol_ProcessLinkState() when a change is detected in the Ethernet link state. The reference versions also update the (sample processor specific) global link state variables, g_phyLinkStatePort1 and g_phyLinkStatePort2. The g_phyLinkStatePort1 and g_phyLinkStatePort2 variables are not needed in the user version but is left intact in the library for testing and reference purposes; it is safe to remove this unneeded code. These functions also enable or disable (respectively) the appropriate port in the fido2100 DLR switch.

To operate properly, BSP_EsLlinkUp() requires the global variable, bsp_EsLinkData, to contain the current Ethernet link speed and duplex information. To help ensure this requirement is met, EtherIpRingProtocol_ProcessLinkState() calls the board support package function BSP_EsUpdateLinkSpeedDuplex() prior to calling BSP_EsLinkUp().

BSP_NotifyLinkStateChange()

The BSP_NotifyLinkStateChange() function is also called by EtherIpRingProtocol_ProcessLinkState() when the Ethernet link state changes. The sample processor specific reference version sets/clears a global link up flag and sends events to the sample processor common interface layer. This function is a hook function that some systems (like a system using the sample processor) may need, but if it is not needed, this function can either be left as an empty stub or the call to it can be safely removed from EtherIpRingProtocol_ ProcessLinkState(). Removal is the choice of the user; this function is left in the library to facilitate testing when used with the sample processor platform.

BSP_EsDumpMediaCount()

The BSP_EsDumpMediaCount() function reads and prints the fido2100 DLR switch counters. It is a debug function and is currently unused. It is implementation independent.

BSP_EsEnablePort() and BSP_EsDisablePort()

The BSP_EsEnablePort() and BSP_EsDisablePort() functions enable or disable (respectively) the specified port in the fido2100 DLR switch. The enable function also indicates the Ethernet link speed and duplex to the fido2100 DLR switch. The DLR library calls BSP_EsEnablePort() (in BSP_EsLinkUp()), but there is no need for the user code to ever call it. BSP_ EsDisablePorts() is also called by the DLR library (in BSP_ EsLinkDown() and BSP_Initialize()), but is called by the user code when the Ethernet/IP Ethernet link object admin state is set to disabled. When the Ethernet/IP Ethernet link object admin state is changed, it is recommended to power up/down (or otherwise enable/disable) the actual PHYs. The powering up and powering down of the PHYs is not managed by the DLR library. When the admin state for a particular port is set to enabled, it is sufficient for the user code to power on (or otherwise enable)

the PHY because the fido2100 DLR switch then detects the link up event and automatically call BSP_EsEnablePort() to enable the respective port in the fido2100 DLR switch. The calling of the BSP_EsEnablePort() is implementation independent.

IMPLEMENTATION INDEPENDENT STATIC FUNCTIONS IN THE .C FILE

bsp_EsStatCountTask()

bsp_EsStatCountTask() is a task that periodically reads the fido2100 DLR switch statistics counters and updates the bsp_EsPortCount global variable. This function is implementation independent.

bsp_EsCalcCrc32()

bsp_EsCalcCrc32() is a helper function to calculate the CRC needed by the fido2100 DLR switch multicast hash table. Only BSP_EsMulticastAdrsAdd uses this function. The helper function is implementation independent.

bsp_EsMcAdrsFind()

bsp_EsMcAdrsFind() is a helper function to search the fido2100 DLR switch multicast hash table for a specific entry. It is used only by BSP_EsMulticastAdrsAdd() and BSP_EsMulticastAdrsDel(). This function is implementation independent.

bsp_EsMcAdrsFindFirstFree()

bsp_EsMcAdrsFindFirstFree() is a helper function to search the fido2100 DLR switch multicast hash table for the first free entry. Only BSP_EsMulticastAdrsAdd() uses this function. This function is implementation independent.

IMPLEMENTATION INDEPENDENT STATIC INLINE FUNCTIONS IN THE .H FILE

Implementation independent static inline functions include the following:

- BSP_EsReadReg16()
- BSP_EsWriteReg16()
- BSP_EsReadReg32()
- BSP_EsWriteReg32()
- BSP_EsReadReg48()
- BSP_EsWriteReg48()
- BSP_EsReadReg64()
- BSP_EsWriteReg64()

All these functions are usable as is; they are implementation independent.

INITIALIZATION

Because it is not desirable for an Ethernet link to be established before the DLR library is ready, the PHYs must be disabled by the user code on power-up. If the chosen PHYs are disabled by default on power-up, no action is required. Most PHYs are enabled until told otherwise, so it is likely that the user code must disable them immediately after the system starts. The library does not manage this aspect of the PHY interface.

As it pertains to the DLR functions, the user initializes the system by taking the following steps:

1. Power up.
2. Consult the PHY data sheet to determine if it is necessary to disable the PHY, and disable the PHY if necessary.
3. Restore the MAC address (and other data as required) from NVM.
4. Initialize the overall system hardware and interrupt subsystem.
5. If using the fido5100BBCZ and fido5200BBCZ REM switch, perform the hardware initialization as described in the REM Switch Driver User Guide.
6. Call EtherIpRingProtocol_Initialize() to initialize the DLR library and the board support package (must also provide proper arguments for DLR enable and switch type).
7. If required, register the DLR library MAC filter add function (BSP_EsMulticastAdrsAdd()) with the Ethernet driver and/or TCP/IP stack.
8. Initialize the pyramid stack by taking the following steps:
    a. Register the Ethernet/IP call back function (clientRegisterEventCallBack()).
    b. Start the Ethernet/IP stack (clientStart()).
    c. Add the assembly instances (clientAddAssemblyInstance()).
    d. Give the Ethernet/IP stack some initial (NULL) assembly data to produce (clientSetAssemblyInstanceData()).
    e. Tell the Ethernet/IP stack to verify connections, (clientSetAppVerifyConnection()).
    f. Get the Identity object from the Ethernet/IP stack (clientGetIdentityInfo()).
    g. Fill in the system specific identity information as appropriate.
    h. Return the identity object to the Ethernet/IP stack (clientSetIdentityInfo()).
    i. Tell the Ethernet/IP stack to support the DLR object (EtherIpRingObject_RegisterObject()).
9. Call EtherIpRingProtocol_SetMACAddress() to tell the hardware, driver, and library what the system MAC address is.
10. Start the DLR library:
    a. Call EtherIpRingProtocol_SetIpAdrs().
    b. Call EtherIpRingProtocol_Start().
11. Power on the PHYs and set the initial admin state (derived from NVM).
12. If necessary, force the Ethernet link speed and duplex.

An Ethernet link can now be established. It is essential that the CPU interrupts and the Ethernet packet receive/transmit capability be in place at this point. When a DLR capable device detects an Ethernet link, it sends a link status packet to the DLR supervisor, and the supervisor immediately sends a sign on packet, so it is essential that the system is capable of responding to DLR packets at this point in the initialization process. It is not necessary to be able to respond to TCP/IP packets because DLR packets are not TCP/IP packets. At this point it is likely that the system does not yet have a valid IP address, especially if using dynamic host configuration protocol (DHCP).

If using DHCP, wait for an IP address before continuing.

If using a static IP address, the user does not need to wait.

Complete the pyramid stack initialization by taking the following steps:

1. Call clientUnclaimAllHostIPAddress()
2. Call clientGetHostIPAddress()
3. Call clientClaimAllHostIPAddress()

INTERRUPTS

All of the fido2100 DLR switch interrupt handler functions are managed by the code in **EtherIpRingProtocol.c**. Interrupts generated by the fido5100BBCZ and fido5200BBCZ REM switch are managed by the REM driver and are not discussed in this user guide. For that information, see the REM Switch Driver User Guide. EtherIpRingProtocol_HandleIrq() is the primary fido2100 DLR switch interrupt handler function. The system interrupt handler that is assigned to the fido2100 DLR switch calls this function. It reads the fido2100 DLR switch event register and routes the particular fido2100 DLR switch interrupt event to the rest of the DLR library. The DLR library handles the interrupt acknowledge required by the fido2100 DLR switch, but it does not acknowledge the underlying physical CPU interrupt.

The fido2100 DLR switch interrupt pin is routed to an available CPU interrupt. The user code must enable, disable, detect, and otherwise manage the physical interrupt interface (vector table address, interrupt acknowledge, and so on) as required. Then the interrupt handler must process the EtherIpRingProtocol_HandleIrq().

The interrupt handler, EtherIpRingProtocol_HandleIrq(), reads the fido2100 DLR switch event register and calls one or more of the following functions:

- EtherIpRingProtocol_HandleLinkStateIrq()
- EtherIpRingProtocol_HandleBeaconTimeoutIrq()
- EtherIpRingProtocol_HandleBeaconReceivedIrq()
- EtherIpRingProtocol_HandleBeaconStateIrq()
- EtherIpRingProtocol_HandlePeriodicTimerIrq()

ETHERNET LINK UP/DOWN

To support the DLR protocol, the detection of the Ethernet link state (up/down) is required. For both the fido2100 DLR switch and the REM switch, the Ethernet link active signal (usually used as a LED output) provided by the PHY must be routed to the link status input signal on the switch. (There are inputs for each port on both switches.) The detection of Ethernet link state is also documented in the REM Switch Driver User Guide. When the link state changes, the signal changes levels, which triggers an interrupt to the CPU. In the case of the fido2100 DLR switch, the CPU interrupt handler function then must acknowledge the CPU interrupt and call the EtherIpRingProtocol_HandleLinkStateIrq() function, which acknowledges the interrupt in the fido2100 DLR switch. Then, read the fido2100 DLR switch event register to determine the link status and call the EtherIpRingProtocol_HandleLinkStateChange() function. In the case of the REM switch, the interrupt handler also must acknowledge the CPU interrupt but can then call the EtherIpRingProtocol_HandleLinkStateChange() function directly. In both cases, this function stores the link state variables and posts, a semaphore that LinkMonTask() is pending on. This task then wakes and calls EtherIpRingProtocol_ProcessLinkState() to complete the processing as described in detail in this section. For link up and link down detection, it is not necessary to interact directly with the PHY. When a link up is detected, it is necessary to determine the link speed and duplex because the Ethernet switch (both types) must be provided with this information. For this purpose, it is required to interact with the PHY. The system must be provided with a method to read the registers in the PHY. This method can be provided by a serial communications protocol using the management data input/output (MDIO) pin and the management data clock (MDC) pin on the PHY. The MDIO pin and MDC pin functionality are not managed by the DLR library.

MODIFYING THE DLR SWITCH SUPPORT LIBRARY FOR USE WITHOUT AN RTOS

INTRODUCTION

The intended use of the fido2100 DLR switch library is in a system that provides some of the features of an RTOS. When an RTOS is not available, it is possible to make some simple modifications to the DLR library (specifically, to the RTOS porting layer) to remove these requirements. This section describes how the RTOS features are used within the library and provides some suggestions to guide the user through modifying the library for use without an RTOS.

The DLR library header file, **BspEnetSwitch.h**, has a list of the RTOS porting layer functions used within the library. These functions are shown in Table 3 along with suggestions for the library modifications and the corresponding application implementation details.

TASK AND THREAD MODIFICATIONS

The Background Information section through the Miscellaneous Modifications section show the modifications that are made to the task and thread related porting layer functions to use the fido2100 support library without an RTOS.

BSP_CreateTask() Porting Layer Function

The BSP_CreateTask() porting layer function is not needed. Replace it with an empty stub.

Table 3. Threads in the DLR Library

Task Function	What Triggers It	What to Change
EtherIpRingProtocol_MainTask()	Pends on a RTOS provided event queue	Use a circular array of integers (events)
EtherIpRingProtocol_LinkMonTask()	Pends on a RTOS provided semaphore	Use a simple Boolean flag
EtherIpRingProtocol_TimerTask()	Pends on a RTOS provided timer	Delete timer and call periodically in main loop
bsp_EsStatCountTask()	Pends on a RTOS provided timer	Delete timer and call periodically in main loop

BACKGROUND INFORMATION

There are only a small number of tasks (threads) used within the DLR library. These tasks are described in Table 3.

In the original DLR library, these tasks are implemented with a standard **while (1)** threading loop, but they can be changed to become simple subroutines by removing that code. After that change, these subroutines are then called by the application main executive loop. The main issue here is to ensure these functions are called on a regular basis and often enough to perform their intended functions.

The first two of these tasks are triggered by functional stimuli (that is, they are not time-based) and can therefore become standard functions called conditionally based on standard variables in memory. The events referred to in this library are a set of enumerations (integers). Therefore, EtherIpRingProtocol_MainTask() can be made to look at integers placed into a simple circular array rather than an RTOS queue. The semaphore triggers EtherIpRingProtocol_LinkMonTask() can become a simple Boolean flag that conditionally calls the resulting subroutine.

Timers provided by an RTOS trigger the other two tasks. Because RTOS timers are not available, it becomes necessary to find another method, but only one of these tasks requires any real timing precision.

For EtherIpRingProtocol_TimerTask(), it is necessary to continue to use a fairly precise timer. The mechanism to create this timer depends on the system resources. The task expects to be called once every 10 ms. If this task is called every 10 ms, there is nothing more to do. If this is not the rate at which this task is called, then it is necessary to modify the library timer macros.

For bsp_EsStatCountTask(), precise timing is not critical. It periodically reads the Ethernet link statistics counters within the fido2100 DLR switch and as long as it is called at least once every 5 sec, these counters are read correctly. Modify this task to remove the **while (1)** thread loop code, delete the existing RTOS timer, and call this periodically (for example, at least once every 5 sec) within the application main executive loop.

The purpose of EtherIpRingProtocol_TimerTask() is to monitor several other DLR library provided timers. These timers are not dependent on an RTOS but are instead completely contained within the DLR library. Other than possibly changing the timer rate, no code changes to these timers are necessary. The timers themselves are merely data structures and are triggered when necessary by the DLR library using the EtherIpRingProtocol_StartTimer() function. When the library uses a timer, it specifies the desired timer delay in milliseconds using a value from a set of timer delay variables (actually macros) in the **EtherIpRingProtocol.c** file.

One time use case is shown by the following code:

```
      #define
NEIGHBOR_CHECK_TIMEOUT_TICKS
((BSP_GetTicksPerSec()/10)/10)
```

This timer is intended to fire after a 100 ms delay. The macro gets the system base tick count using BSP_GetTicksPerSec() and then divides that by 10 to get the base timer count for 100 ms. Then, this result is divided again by 10, because EtherIpRingProtocol_TimerTask() examines the timers once every 10 ms. The other macros are similar, but all require the timer task to be called at the expected rate.

For instance,

> BSP_GetTicksPerSec()/10 = (system ticks per second)/(time fires per second)

Calculate as follows:

System Ticks/Timer Fire

> System Ticks per Timer Fire/10 = ?
>
> (System Ticks per Timer Fire)/(System Ticks per Timer Interval) = Timer Intervals per Timer Fire

This number is the number of timer task iterations before this particular timer is triggered.

Relatively long timer delays can also be defined by the following code; #define yields a 30 second delay:

```
      #define
RING_RAPID_FAULTS_TIMEOUT_TICKS
((BSP_GetTicksPerSec()*30)/10)
```

SEMAPHORE MODIFICATIONS

In the modified library for use without an RTOS, the BSP_SemaphoreInit() section through the Background Information section show the modifications that are made to the semaphore related porting layer functions to use the fido2100 support library without an RTOS.

BSP_SemaphoreInit() Porting Layer Function

The BSP_SemaphoreInit() porting layer function is no longer needed. Replace it with an empty stub.

BSP_SemaphoreWait() Porting Layer Function

Replace the BSP_SemaphoreWait() porting layer function with an equivalent function that accomplishes the following tasks:

- Disable interrupts
- Examine flag (and clear it if set)
- Reenable interrupts
- Return flag

BSP_SemaphorePost() Porting Layer Function

Replace the BSP_SemaphorePost() porting layer function with an equivalent function that accomplishes the following tasks:

- Disable interrupts
- Set flag
- Reenable interrupts
- Return

Background Information

There is only a single semaphore used in the DLR library: linkMonTaskSem. It is used to wake EtherIpRingProtocol_LinkMonTask(). If that task is changed into a main loop subroutine, a flag can be used in place of the semaphore to conditionally call the subroutine. The only caution is that this semaphore is posted in EtherIpRingProtocol_HandleLinkStateChange(), which is usually called in an application provided interrupt handler. Take care to ensure the flag is protected from simultaneous access by disabling interrupts around the code that sets the flag.

MUTEX MODIFICATIONS

The following sections show the modifications that are made to the mutex related porting layer functions to use the fido2100 support library without an RTOS.

BSP_MutexInit() Porting Layer Function

In the modified mutex, the BSP_MutexInit() porting layer function is no longer needed. Replace it with an empty stub.

BSP_MutexLock() Porting Layer Function

In the modified mutex, the BSP_MutexLock() porting layer function is no longer needed. Replace it with an empty stub.

BSP_MutexUnlock() Porting Layer Function

In the modified mutex, the BSP_MutexUnlock() porting layer function is no longer needed. Replace it with an empty stub.

Background Information

There is only one mutex used in the DLR library: timerMutex. It protects the m_tmrList array. The m_tmrList array is a set of data structures that manages the DLR library internal timers. These timers keep track of the various time related DLR protocol functions. If the EtherIpRingProtocol_TimerTask is converted into a main loop subroutine, then less sophisticated protection is needed. Most likely, all that is required is to disable interrupts around the access to these data structures within EtherIpRingProtocol_TimerTask().

EVENT MODIFICATIONS

The BSP_GetEvent() section through the Background Information section show the modifications that are made to the event related porting layer functions to use the fido2100 support library without an RTOS.

BSP_GetEvent() Porting Layer Function

Replace the BSP_GetEvent() porting layer function with an equivalent function that accomplishes the following tasks:

- If the event array is not empty, do the following:
  - Disable interrupts
  - Get an event ID from event array
  - If necessary, circularize the array
  - Reenable interrupts
  - Return event ID
- If the event analysis is empty, do the following:
  - Return zero

BSP_PostEvent() Porting Layer Function

Replace the BSP_PostEvent() porting layer function with an equivalent function that accomplishes the following tasks:

- Disable interrupts
- Place provided event ID into event array
- Circularize the array if necessary
- Reenable interrupts
- Return

Background Information

In the DLR library, events trigger EtherIpRingProtocol_MainTask() to perform various operations. An event is an integer ID code that indicates an event type. (A list of the defined event types can be found in **EtherIpRingProtocol.c.**) When not using an RTOS, the event queue can become a simple, circular array that holds these integers. An array size of 16 integers is sufficient. After this change is made, the get and post functions can become subroutines to access this array (as a FIFO) and to keep it circular.

TIMER MODIFICATIONS

The BSP_TimerCreate() section through the Background Information section show the modifications that are made to the timer related porting layer functions to use the fido2100 support library without an RTOS.

BSP_TimerCreate() Porting Layer Function

In the modified timer, the BSP_TimerCreate() porting layer function is no longer needed. Replace it with an empty stub.

BSP_TimerWait() Porting Layer Function

In the modified timer, the BSP_TimerWait() porting layer function is no longer needed. Replace it with an empty stub.

Background Information

There is a pair of RTOS provided timers used in the DLR library: one is in EtherIpRingProtocol_TimerTask(), and the other is in bsp_EsStatCountTask(). Because both timers are used to pace these tasks, and the tasks are to be converted to periodic subroutine calls in the main loop executive, there is no longer any need for these RTOS timers; they can be removed.

The DLR library internal timers (see the wdTimer struct) are distinct from the RTOS provided system timers. These timers are completely self-contained within the library. In other words, the timers are merely an array of data structures that are scanned periodically to trigger various events on a timed basis.

MISCELLANEOUS MODIFICATIONS

BSP_GetTicksPerSec() Porting Layer Function

Ensure the BSP_GetTicksPerSec() function returns an appropriate number for the given system architecture.

BSP_EsEnablePort()Porting Layer Function

The BSP_EsEnablePort() function is unaffected. No changes are needed.

BSP_EsDisablePort()Porting Layer Function

The BSP_EsDisablePort() function is unaffected. No changes are needed.

Background Information

Strictly speaking, these functions are not RTOS related. The BSP_GetTicksPerSec() function returns a numeric constant that represents the rate of the system timer tick. It is a requirement to have some sort of periodic timer that sets a system timer tick value. Usually, the implementation of the periodic timer is a simple hardware timer triggering an interrupt on a periodic basis. This function returns the ticks per second rate of this timer tick. The other functions do not require any changes.